Lecture notes for CS 6110 (Spring'09) taught by Andrew Myers at Cornell; edited by Amal Ahmed, Fall'09.

1 Strong normalization

In previous lectures, we have seen that in the simply-typed lambda calculus ($\lambda^{\rightarrow}$) we cannot write down a well-typed version of the (untyped) term Ω. In $\lambda^{\rightarrow}$, all programs terminate. In fact, it doesn't matter which evaluation strategy we use: $\lambda^{\rightarrow}$ programs terminate in normal forms under any reasonable evaluation strategy.

Evaluation is weakly normalizing if all values (i.e., irreducible terms) reachable by evaluation are equivalent, i.e., they are the same normal form. But it doesn't guarantee that all (or any) evaluations reach a value. Evaluation is strongly normalizing if all evaluations reach a normal form.

We'll prove the property of strong normalization for CBV evaluation. We use the following abbreviations:

$$\begin{array}{rcl} e \Downarrow v & \overset{\text{def}}{=} & e \longrightarrow^* v \\ e \Downarrow & \overset{\text{def}}{=} & \exists v. e \Downarrow v \end{array}$$

That is, we write $e \Downarrow v$ iff e evaluates to a value v in zero or more steps under the small-step CBV operational semantics. We write $e \Downarrow$ when e evaluates to some value (i.e., when evaluation of e terminates).

Since CBV evaluation is deterministic, we know that e converges iff e does not diverge: $e \Downarrow \Longleftrightarrow e \Uparrow$. If e both converged and diverged, then the convergent evaluation of e could reach some value v. Determinism of evaluation implies that any divergent evaluation from e would be alpha-equivalent at every step to the convergent evaluation. But any term alpha-equivalent to v has to be a value itself, and could not take any additional step.

For our deterministic CBV calculus, we can express strong normalization as:

$$\vdash e : \tau \Longrightarrow e \Downarrow$$

To prove this, we introduce a new proof technique, *logical relations*. In this technique, we define a relation over terms, where the relation is indexed by a type and is defined by structural induction on that type. For the purposes of of this proof, we define a *unary* logical relation SN_τ . A unary relation is just a set, so we write $SN_\tau(e)$ to mean that e is a member of the set for the type τ. The definition of SN_τ has three kinds of clauses for each kind of τ:

1. The condition that e has type τ, that is, $\vdash e : \tau$.
2. The condition we wish to prove, $e \Downarrow$.
3. A condition that enables us to prove that the logical relation is preserved by evaluation of elimination forms for type τ.

For the simple case of $\lambda^{\rightarrow}$, we can define SN_τ as follows by structural induction on τ. (Below, B ranges over base types, while b ranges over constants of base type.)

$$\begin{array}{rcl} SN_B(e) & \Longleftrightarrow & \vdash e : B \ \wedge \ e \Downarrow \\ SN_{\tau_1 \to \tau_2}(e) & \Longleftrightarrow & \vdash e : \tau_1 \to \tau_2 \ \wedge \ e \Downarrow \\ & & \wedge \ \forall e'. \ SN_{\tau_1}(e') \Longrightarrow SN_{\tau_2}(e\ e') \end{array}$$

The final clause of the definition of $SN_{\tau_1 \to \tau_2}$ corresponds to (3) above. Note that although it is defined in terms of a universal quantification over e', the definition is well-founded because $SN_{\tau_1 \to \tau_2}$ is defined in terms of SN_{τ_1} and SN_{τ_2}, and τ_1 and τ_2 are smaller types than $\tau_1 \to \tau_2$.

2 Some properties of the logical relation

We can now state some important lemmas.

Lemma 1

$$SN_\tau(e) \Longrightarrow e \Downarrow$$

This is obvious from the definition. In fact, while the name *SN* is suggestive of "strong normalization", the property is stronger, because of clause (3).

Lemma 2

$$\vdash e : \tau \wedge e \longrightarrow e' \wedge SN_\tau(e') \implies SN_\tau(e) \qquad \textit{(2a)}$$
$$\vdash e : \tau \wedge e \longrightarrow e' \wedge SN_\tau(e) \implies SN_\tau(e') \qquad \textit{(2b)}$$

This lemma says that the SN_τ property is preserved when we walk either backward or forward in the evaluation sequence. The proof of both parts is similar, so we show just the first part (2a).

Proof: by structural induction on τ. In each case we assume $\vdash e : \tau \wedge e \longrightarrow e' \wedge SN_\tau(e')$, and show $SN_\tau(e')$.

Case $\tau = B$: If we have $SN_B(e')$, then e' converges. But since $e \longrightarrow e'$, then e converges too. From $\vdash e : B$ and $e \Downarrow$, we conclude $SN_B(e)$.

Case $\tau = \tau_1 \to \tau_2$: As in the previous case, we have $e \Downarrow$. We also need to show $SN_{\tau_1}(e'') \implies SN_{\tau_2}(e\ e'')$ for an arbitrary e''. Consider such e''. We have $SN_{\tau_1\to\tau_2}(e')$, so from its definition, we know $SN_{\tau_2}(e'\ e'')$. Since $e \longrightarrow e'$, we also know that $e\ e'' \longrightarrow e'\ e''$ from the CBV evaluation rules. Since $\tau_2 \prec \tau_1 \to \tau_2$, we can apply the induction hypothesis to $e\ e''$, obtaining $SN_{\tau_2}(e\ e'')$, as desired.

We need one more lemma that lets us do substitutions. The reason is that strong normalization is a property of closed terms, but because we construct a proof by induction on typing derivations, we need to consider open terms (the typing rule for lambda abstractions involves typing the function body, which is open in general). However, we can close open terms by performing a substitution that replaces all free variables with terms.

Let γ be a *finite substitution*, that maps from variables to values, e.g. $\gamma = \{x_1 \mapsto v_1 \ldots x_n \mapsto v_n\}$. We say that γ satisfies a typing context Γ, written $\gamma \models \Gamma$, if both have the same domain, and γ maps variables onto values that are of the right type $\Gamma(x)$ and that also satisfy the *SN* property at that type:

$$\gamma \models \Gamma \iff \mathrm{dom}(\gamma) = \mathrm{dom}(\Gamma) \wedge \forall x \in \mathrm{dom}(\gamma).\ SN_{\Gamma(x)}(\gamma(x))$$

We write $\gamma(e)$ to mean the substitution in e of all variables in the domain of γ with the corresponding values:

$$\gamma(e) = e\{v_1/x_1\} \ldots \{v_n/x_n\}$$

We need a substitution lemma regarding finite substitutions:

Lemma 3

$$\Gamma \vdash x : \tau \wedge \gamma \models \Gamma \implies \vdash \gamma(x) : \tau$$

Proof: This is proved by induction on the size of the domain of γ. The case $n = 1$ is exactly the substitution lemma that we used to prove Preservation. And that same lemma can be used to prove the induction step.

With these definitions, we can now prove the main result:

Lemma 4

$$\Gamma \vdash e : \tau \wedge \gamma \models \Gamma \implies SN_\tau(\gamma(e))$$

Notice that if we instantiate this with $\gamma = \emptyset$, $\Gamma = \emptyset$, then we get $\vdash e : \tau \implies SN_\tau(e)$, which implies strong normalization by Lemma 1.

We prove Lemma 4 by induction on the typing derivation $\Gamma \vdash e : \tau$.

- Case $\Gamma \vdash b : B$. Since $b = \gamma(b)$, clearly $\gamma(b) \Downarrow$ and $\vdash \gamma(b) : B$. Therefore, we know $SN_B(\gamma(b))$.
- Case $\Gamma \vdash x : \tau$. It must be the case that $\Gamma(x) = \tau$, and because $\gamma \models \Gamma$, therefore $SN_\tau(\gamma(x))$, as required.

- Case $\Gamma \vdash e_0\ e_1 : \tau$. We know from the typing derivation that the premises $\Gamma \vdash e_0 : \tau_1 \rightarrow \tau$ and $\Gamma \vdash e_1 : \tau_1$ hold for some type τ_1. We apply the induction hypothesis to get $SN_{\tau_1\rightarrow\tau}(\gamma(e_0))$ and $SN_{\tau_1}(\gamma(e_1))$. From the definition of $SN_{\tau_1\rightarrow\tau}$ (clause 3), this implies $SN_\tau(\gamma(e_0)\ \gamma(e_1))$. But by the definition of substitution, this is the same as $SN_\tau(\gamma(e_0\ e_1))$.

  Notice that without that third clause (which we were able to introduce as part of the definition of the logical relation), we would have been stuck at this point if we had just tried to prove the theorem directly by induction on the typing derivation.

- Case $\Gamma \vdash \lambda x\!:\!\tau_1.\, e_2 : \tau_1 \rightarrow \tau_2$. This is the only tricky case, because we need to prove the third clause that we exploited in the application case. We need to show $SN_{\tau_1\rightarrow\tau_2}(\gamma(e))$. This requires proving three clauses.

  The first clause requires that $\gamma(e)$ has the right type. This comes trivially from the typing derivation and Lemma 3.

  The second clause requires that $\gamma(e)$ converges. Since γ maps variables only to values, there is no possibility of variable capture in the substitution $\gamma(e)$. Therefore, $\gamma(e) = \lambda x\!:\!\tau_1.\,(\gamma\backslash x)(e_2)$, where $\gamma\backslash x$ is the same as γ, without any mapping for x. Since $\gamma(e)$ is a value already, the second clause is also trivial.

  The third clause requires that for an arbitrary e' satisfying $SN_{\tau_1}(e')$, we have $SN_{\tau_2}(\gamma(e)\ e')$. Consider such an e'. How does the term $\gamma(e)\ e'$ evaluate? Since $\gamma(e)$ is already a value, the right-hand side (e') evaluates until it reaches a value. Since we assumed $SN_{\tau_1}(e')$, its evaluation reaches some value v' by Lemma 1. By Lemma 2b, the value v' satisfies $SN_{\tau_1}(v')$. The next step is to substitute v' for x in the function body: $\gamma(e)\ v' \longrightarrow (\gamma\backslash x)(e_2)\{v'/x\}$. But we can fold the substitution for x into γ, making this $\gamma[x \mapsto v'](e_2)$.

  From the typing derivation for e, we know $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. If $\gamma \models \Gamma$, then $\gamma[x \mapsto v'] \models \Gamma, x : \tau_1$. So we can use the induction hypothesis to conclude $SN_{\tau_2}(\gamma[x \mapsto v'](e_2))$. Since $\gamma(e)\ e'$ steps to this term in a finite number of steps, we can conclude by induction on the number of steps (and Lemma 2a) that $SN_{\tau_2}(\gamma(e)\ e')$, as required.

3 Discussion

The technique of logical relations generalizes to more expressive languages. We'll shortly see extensions of the lambda calculus that can be used to write more interesting computations, yet can be proved strongly normalizing with the same technique.

And there are situations in which it is useful to have a language in which all programs terminate. For example, operating systems and web browsers are often extended with plug-in software that is not fully trusted. Knowing that the plug-in code can't create an infinite loop is useful (though we probably want an even tighter bound on run time). Also, we'll later see type systems with type expressions isomorphic to the lambda calculus (parameterized types). Because evaluation in the type language terminates, the type checker also terminates, which is a useful property!

Extending the proof of strong normalizaton

To extend the proof of strong normalization to a language with **Unit**, **Bool**, and product types, we define the logical relation as follows:

$$\begin{aligned}
SN_{\mathbf{Unit}}(e) &\iff \vdash e : \mathbf{Unit} \;\wedge\; e \Downarrow \\[1em]
SN_{\mathbf{Bool}}(e) &\iff \vdash e : \mathbf{Bool} \;\wedge\; (\exists v.\ e \Downarrow v \\
&\qquad \wedge\; (v = \mathbf{true} \;\vee\; v = \mathbf{false})) \\[1em]
SN_{\tau_1 \to \tau_2}(e) &\iff \vdash e : \tau_1 \to \tau_2 \;\wedge\; e \Downarrow \\
&\qquad \wedge\; \forall e'.\ SN_{\tau_1}(e') \Longrightarrow SN_{\tau_2}(e\ e') \\[1em]
SN_{\tau_1 \times \tau_2}(e) &\iff \vdash e : \tau_1 \times \tau_2 \;\wedge\; e \Downarrow \\
&\qquad \wedge\; SN_{\tau_1}(\mathbf{fst}\ e) \;\wedge\; SN_{\tau_2}(\mathbf{snd}\ e)
\end{aligned}$$

Note that for product types, we can alternatively define the logical relation as follows. (Hint: You may wish to refer to the version below as you try to define the logical relation for sum types.)

$$\begin{aligned}
SN_{\tau_1 \times \tau_2}(e) &\iff \vdash e : \tau_1 \times \tau_2 \;\wedge\; (\exists v_1, v_2.\ e \Downarrow (v_1,\, v_2) \\
&\qquad \wedge\; SN_{\tau_1}(v_1) \;\wedge\; SN_{\tau_2}(v_2))
\end{aligned}$$